DSA Homework 6: Hash Maps, Proofs, and Trees

David Tarazi

March 5, 2020

1 Sick Patients in O(n) Time

In preprocessing, I will loop through all of the students at Olin and create two hash maps. In the first hash map, A, the keys are the names of all students, and the values hold a list of all the classes they are in. Then, in hash map B, the keys are the names of all students and the values hold either positive or negative for academitis, but are initialized to negative. Using these two hash maps, we can then find out if a student potentially has the disease.

To start the actual function, I will create a queue that holds all of the unchecked students and a list of the suspected students, then enqueue patient 0. Next, I will create a loop that continues until the queue is empty. Within each loop iteration, I will dequeue that student from the queue of students we need to check, and then loop through all of that students classmates (a finite number) which was retreived in O(1) time from hash map A. For each student, I look at hash map B and check if that student is negative. If they are negative, I add them to the suspected student list and enqueue that student to the unchecked list. Finally, I update the status in hashmap B so that I don't check a student more than once. If they are positive, I leave them so that I don't check duplicates. The while loop runs in O(n) time and within each loop I check 120 classmates. Since lookup time is O(1) with a hash map, the time is O(120n) = O(n).

Proof by Contradiction: Suppose a student caught academitis, but wasn't added to the suspected student list. In the instance that this happened, either that student was not marked positive, or was not added to the queue. However, the algorithm does not allow for that to happen as we check each patient and always update their status and add to the list if they are negative and in contact with anyone who was positive, creating a contradiction.

2 Proof by Induction: Merge Sort

Inductive Hypothesis: At every level i of rebuilding the sorted list after broken down into individual elements, the list is sorted.
Base Case: At layer i, each element has been broken down into a sublist of length 1, meaning each sublist is sorted.
Level i+1: Suppose we are merging two sorted lists at layer i+1, and this continues until we have combined the sublists back into a single sorted list. At this layer we are merging two sorted lists and we know that the first half of the combined list is sorted and likewise, the second half is also sorted, but independently. At each layer, we utilize insertion sort on the second half of the list to fully sort the list. Therefore, we take two independently sorted lists and combine them into a new sorted list at each layer until we have a single sorted list.

3 Union-Find

Using doubly-linked lists, you can have each node contain the normal next and previous pointers, but you can also include a pointer to the head of the set in each node and the length of the set. Thus, for creating a set, I initiate a head node that stores None as the head since it is its own head, None for previous and next

since it stands alone, and a length of 1. This functionality runs in O(1) time. For union(A,B), I check which set has the smaller length and I loop connect the tail of the larger set to the head of the smaller set and then loop through the smaller set to update the head to point to the smaller set's head instead. Therefore, I am iterating through the smaller list's elements and doing an O(1) operation each time, making this function O(min(len(A), len(B))) time. Finally to get the head using find(i), I can return the value of the head pointer from any index in a set in O(1) time.

Using a tree, when making a set, the first element is the head/root of the tree. This node has no head beyond itself and it has its own value and runs in O(1) time. When creating a union, the branch stems from A and goes to B and then B's head is a pointer to A. Therefore, if you wanted to run all the way back to the original root, you could follow head pointers eventually back there, but the immediate parent is only one step away. This action takes O(1) time since you only need to add one child node and update one head pointer. Lastly, to find the head given some index, i, you can follow the parents head pointers to the root, but the immediate head is just one step away. The timing of this is O(1) to find the immediate head, but for the root, it depends on how many unions there are but the worst case is a connection in the tree where each parent has one child and it would be O(n). Best case would be one parent having all the children which would be O(1) time. ** I couldn't figure out how to do just O(logn) time.